WMM: a Resilient Weak Memory Model

by

Sizhuo Zhang

B.E., Tsinghua University (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2016

Author .....
Department of Electrical Engineering and Computer Science
January 27, 2016

Certified by .....
Arvind
Johnson Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

WMM: a Resilient Weak Memory Model

by

Sizhuo Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on January 27, 2016, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

A good memory model should have a precise definition that can be understood by any computer architect readily. It should also be *resilient* in the sense that it should not break when new microarchitecture optimizations are introduced to improve single-threaded performance. We introduce WMM, a new weak memory model, which meets these criteria. WMM permits all load-store reorderings except a store is not allowed to overtake a load. WMM also permits both *memory dependency speculation* and *load-value prediction*. We define the operational semantics of WMM using a novel conceptual device called *invalidation buffer*, which achieves the effect of out-of-order instruction execution even when instructions are executed in-order and one-at-a-time. We show via examples where memory fences need to be inserted for different programming paradigms. We highlight the differences between WMM and other weak memory models including Release Consistency and Power. Our preliminary performance evaluation using the SPLASH benchmarks shows that WMM implementation performs significantly better than the aggressive implementations of SC. WMM holds the promise to be a vendor-independent stable memory model which will not stifle microarchitectural innovations.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science and Engineering

Acknowledgments

I would like to thank my advisor, Prof. Arvind, for his support and guidance that made this project possible. I also want to thank Prof. Daniel Sanchez for his help and advice on simulation experiments. Finally I want to acknowledge Muralidaran Vijayaraghavan for discussing many technical details with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The importance of memory models is difficult to deny; how can one specify an Instruction Set Architecture (ISA) if the meaning of load (Ld) and store (St) instructions is not precise. How can one give the semantics of a multithreaded programming language without specifying the behavior of shared writeable variables? Yet it is a topic that most architecture and language researchers, except those who work on the topic, want to avoid. Purely from a pragmatic point of view if the programmer or the compiler writer totally ignores the memory model issues, he or she is unlikely to pay a price for it; memory-model bugs are ephemeral and other bugs in parallel programming, such as races and deadlocks, manifest themselves much more readily [32]. Further to attribute a bug to the violation of a memory model is problematic because precise and understandable definitions of memory models are lacking [6, 42, 33]. This thesis presents WMM, a new weak memory model, with the goal of providing a vendor-independent memory model which is easy to understand and which is resilient to microarchitectural innovations.

It is important to recognize on the onset that unlike ISA, memory models were never "designed" by architects. Every ISA contains Ld and St instructions whose meaning is quite obvious in a uniprocessor setting or for a single-threaded program; Ld a returns the value stored by the most recent store to the address a. Every optimization in a uniprocessor preserves this abstraction. Instructions can be executed speculatively and out of order, store buffers and caches can be introduced, as long as

the data dependencies are observed by the Ld and St instructions in a thread. None of these architectural mechanisms are visible to the user program. Another condition imposed by almost all general purpose ISAs is that the interrupts are *precise*, i.e. instructions are retired in order even if they are executed out of order. Precise interrupts are needed for the implementation of virtual memory and many other operating system services. It would be pretty messy to implement a system if it allowed stores to be retired out of order.

Unfortunately, the simple load-store abstraction breaks down in a system where multiple threads share a common global memory because a load in a thread can read the value written by a store in some other thread. So it is no longer sufficient to talk about simple data dependencies within a single thread. The earliest specification of a memory model was Sequential Consistency (SC) which specified how threads were allowed to interact with each other. SC specified that the program behavior must appear as if the instructions of various threads were executed one-by-one in an interleaved manner. It was clear from the earliest days that enforcing SC at the microarchitecture level downgraded performance unless aggressive speculation techniques requiring significant extra hardware cost were employed [22, 40, 27, 24, 17, 48, 14, 45, 30, 26]. Worse yet, some of this cost had to be paid even when a program ran in a single-threaded mode. Currently most manufacturers expose weaker memory models than SC and provide instructions to enforce SC as needed. In spite of all the advances in implementation, the performance advantage of weaker models over SC is enough that the manufacturers are unlikely to give up on weaker memory models any time soon.

Ideally we want a memory model to have the following properties:

1. Resilience: At the implementation level, the model should permit microarchitectural optimizations such as out-of-order execution, memory dependency speculation and load-value prediction [31, 21, 39, 38].

2. Simple description: The description of Ld and St instructions should be specified in terms of *instantaneous instruction execution* (I^2E) using monolithic multi-

ported memory. So far such descriptions exist only for SC and Total Store Order (TSO).

3. Vendor independent: The memory model, like SC, should not depend upon the peculiarities of an ISA.

4. Completeness: Model should include *read-modify-write*, *memory fences* and other memory instructions needed to write parallel programs.

1.1 Summary of Contributions

The main contributions of this thesis are:

1. WMM, a new memory model that meets the above goals;

2. A novel technique for describing out-of-order execution using a conceptual device called *invalidation buffer* which achieves the effect of out-of-order instruction execution even when instructions are executed in-order;

3. Formal separation of memory model and cache coherence issues using *purely coherent memory* (PCM) abstraction;

4. A preliminary quantitative evaluation that shows that WMM implementation performs up to 33% and on average 14% better than the aggressive implementation of SC.

1.2 Thesis Organization

In Chapter 2 we discuss other weak memory models and the techniques used to describe them. We also survey aggressive implementations of SC. In Chapter 3 we explain Instantaneous Instruction Execution (I^2E) and use it to describe SC and TSO. WMM is introduced in Chapter 4 where we give its operational semantics, and explain it further via programming examples. In Chapter 5, we show that WMM semantics are not violated even under very aggressive out-of-order and speculative

implementations. In Chapter 6 we separate memory model issues from cache coherence. In Chapter 7 we analytically compare WMM against other weak memory models including Release Consistency and Power. In Chapter 8 we present our preliminary quantitative evaluation of WMM implementation on SPLASH-2x benchmarks [49, 1, 2] and contrast its performance with SC and TSO implementations with and without store prefetch. We finally offer some conclusions in Chapter 9.

Chapter 2

Related Work

2.1 Sequential Consistency

Sequential Consistency (SC) [29] is obviously the most intuitive memory model and has been used since nineteen sixties, even before the existence of multiprocessors (see Dijkstra [19]). However naive implementations of SC suffer from poor performance because the strong instruction ordering required by SC invalidates almost all optimizations for uniprocessor designs. Gharachorloo et al. [22] proposed load speculation and store prefetch to enhance the performance of SC. As the understanding of out-of-order and speculative microarchitectures has improved over the years, researchers have proposed more and more aggressive techniques to preserve SC [40, 27, 24, 17, 48, 14, 45, 30, 26]. Perhaps because of their hardware complexity and performance gap, no commercial microprocessor has adopted these techniques. Manufacturers have chosen instead to present a weaker memory model than SC as the memory interface.

2.2 Weak Memory Models

During the nineteen nineties a plethora of weak memory models emerged to characterize the memory systems of multiprocessor systems. SPARC architecture manual [46] specified Total Store Order (TSO), which relaxed the ordering between a younger

load and an older store, and Partial Store Order (PSO), which further relaxed the ordering between two stores. Goodman [25] proposed Processor Consistency (PC), in which "the order of stores by two processors as observed by them and by a third processor may be different". Dubious et al. [20] defined Weak Consistency (WC), which required the shared variables in a program to be classified into synchronizing variables and non-synchronizing variables, and it only enforced ordering with respect to accesses to synchronizing variables. Gharachorloo et al. [23] presented Release Consistency (RC), a model closely related to WC but in which synchronization accesses were further partitioned into *acquire* and *release* accesses. Based on the type of the synchronization access (i.e. acquire or release), RC can further relax the ordering between the synchronization access and some other accesses. The memory models of Power [28] and ARM [11] as specified in the manuals were similar to WC, but orderings were enforced by fences (instead of synchronization accesses). The manual stated additional conditions such as "dependent loads will not be reordered", and such conditions have turned out to be both vague and imprecise. The tutorial by Adve et al. [3] and the tutorial by Maranget et al. [36] provide introductions to and relationships among above models.

2.3 Describing Memory Models

The lack of clarity in the definition of weak memory models has generated two types of research efforts to fix the problem. One type of effort has tried to develop a detailed *axiomatic semantics* to characterize a specific commercial memory model accurately [6, 8, 5, 33, 9]. Although the accuracy of the models has improved in the sense it conforms to empirical observations, often the models themselves have become too complicated to reason about. The second type of effort has been to describe the model by specifying its *operational semantics* [43, 42, 41, 7]. Namely, the model itself is described as an abstract machine, and the legal behaviors of the memory model had to be observable by executing programs on this abstract machine. The success of this approach depends upon the simplicity of the abstract machine. For

example, the abstract machine for x86 includes a store buffer and results in an easy-to-understand semantics [43]. In fact, Owens et al. prove the equivalence of the axiomatic and operational semantics of TSO [37]. We also define the semantics of WMM in a similar way by introducing a novel conceptual device called invalidation buffer.

Instead of describing memory models directly, Adve et al. [4] define a synchronization model called Data-Race-Free-0 (DRF0), which software programs should obey. In their proposal, the behavior of such programs is contained in SC. However, the model cannot specify the behavior of programs that do not obey DRF0. Our semantics do not require any labelling of memory access and is valid for all programs including the pathological ones.

A large amount of research has also been devoted to specifying and formalizing the memory models of high level languages: C++ [15, 13], Java [35, 16, 34], etc. This remains an active area of research because a widely accepted memory model for high-level parallel programming is yet to emerge.

There are also several other proposals on weak memory models: Shen et al. [44] introduce a model called Commit-Reconcile & Fences (CRF), in which loads and stores affect only the local cache, and Commit and Reconcile instructions are used to control the movement of values between local caches and the global memory. Arvind and Maessen [12] propose a weak memory model which is a combination of instruction reordering and monolithic memory. It specifies precise conditions for preserving store atomicity in program execution even when instruction reordering is permitted. In contrast, the WMM model presented in this thesis does not insist on the atomicity of stores at the program level.

Chapter 3

Defining Memory Models

We will model multiprocessor systems as shown in Figure 3-1 to define memory models. The state of the system with n processors is defined as $\langle ps, m \rangle$, where m is an n-ported *monolithic* memory which is connected to the n processors and $ps[i]$ $(i = 1 \ldots n)$ represents the state of the i^{th} processor. Each processor contains a register state s, which represents all architectural registers, including both the general purpose registers and special purpose registers, such as PC. Specific memory models may add additional state elements, e.g. a *store buffer*, to each processor.

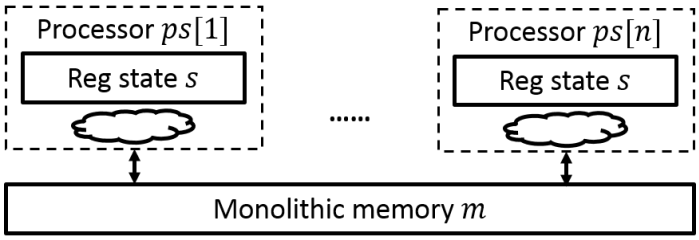

Figure 3-1: General model structure

3.1 Abstracting the Instruction Set

Memory model is always part of the ISA. However, we want our definitions of the memory models to be as generic as possible. For this reason, we introduce the concept of *decoded instruction set* (DIS). A *decoded instruction* contains all the information

of an instruction after it has been decoded and has read all source registers. To begin with our DIS has the following three instructions.

- $\langle \mathsf{Nm}, op, regs \rangle$: instructions that do not access memory, such as ALU or branch instructions. op is the type of operation performed by the instruction, and $regs$ represents all the necessary source register values and destination register names.

- $\langle \mathsf{Ld}, a, dst \rangle$: a load that reads memory address a and updates the destination register dst.

- $\langle \mathsf{St}, a, v \rangle$: a store that writes value v to memory address a.

Later we will extend the DIS with fence and atomic read-modify-write instructions as needed.

Next we explain how we get to decoded instructions from the source or raw instructions.

3.1.1 Instantaneous Instruction Execution ($\mathbf{I^2E}$)

To define memory models we restrict ourselves to the $\mathrm{I^2E}$ model where each instruction is executed instantaneously and the register state of each processor is by definition always up-to-date. Therefore we can define the following two methods to manipulate the register state s of a processor:

- $\mathsf{decode}()$: fetches the next raw instruction and returns the corresponding decoded instruction based on the current register state s.

- $\mathsf{execute}(dIns, ldRes)$: updates the register state s (e.g. by writing destination registers and incrementing PC) according to the current decoded instruction $dIns$. The Ld instruction requires a second argument $ldRes$ which should be the loaded value.

One limitation of $\mathrm{I^2E}$ is that it cannot be used to describe the semantics of an instruction set where the meaning of an instruction may depend upon a future store

instruction. The memory models we discuss do not permit stores to overtake loads in execution.

After introducing our notation for operational semantics, we will give I^{2}E definitions of SC and TSO, and follow it by introducing our weak memory model WMM in Chapter 4.

3.2 Operational Semantics

The operational semantics is a set of *rules* that describe how the state of the processor and memory evolves as execution progresses. Each rule takes the following form:

$$\frac{predicates\ (based\ on\ the\ current\ state)}{actions\ (to\ modify\ the\ current\ state)}$$

If all *predicates* of a rule are *satisfied* then it can *fire* and atomically update model states according to the specified *actions*.

A predicate is either a *when* statement or a *pattern matching* statement. For example, $\mathsf{when}(b.\mathsf{empty}())$ means that the rule requires buffer b to be empty in order to fire. The *pattern matching* statement has the following form:

$$pattern = expression$$

For example, if we want to match the instruction returned by the $\mathsf{decode}()$ method to be a Nm instruction, we can write $\langle \mathsf{Nm}, op, regs \rangle = ps[i].s.\mathsf{decode}()$. Free variables op and $regs$ will be assigned to appropriate values if the matching is successful.

We use "$\Leftarrow$" to assign a new value to a state, and use semicolon ";" to separate statements written on the same line. If multiple rules can fire then our semantic model selects any one of those rules to execute. The final outcome may depend on the choice of rule selection, i.e. the rules are not necessarily "confluent".

To better understand the notation introduced above, we use it to specify two well-known strong memory models, i.e. SC and TSO.

3.3 SC Model

In SC model, a processor contains only the register state s. The operational semantics of SC is shown in Figure 3-2. The three rules correspond to the *instantaneous execution* of the three types of decoded instructions. In each rule, the decode() method first fetches and decodes a new instruction, and then the instruction is immediately executed and committed. Loads and stores in SC directly access the monolithic memory. For example, the SC-Ld rule executes a load by reading the monolithic memory. The three rules are disjoint, i.e. only one of them can be ready to fire. However, in a multi-processor setting there is still a choice regarding which processor we select for execution. The order of firing rules gives a total order of all loads and stores that is consistent with the program order on each processor. Even though SC permits different global Load-Store reorderings, it can be shown formally that reordering loads and stores on a single processor can take us out of the set of permitted behaviors.

SC-Nm rule (Nm execution).

$$\frac{\langle \mathsf{Nm}, op, regs\rangle = ps[i].\mathsf{decode}()}{ps[i].s.\mathsf{execute}(\langle \mathsf{Nm}, op, regs\rangle)}$$

SC-Ld rule (Ld execution).

$$\frac{\langle \mathsf{Ld}, a, dst\rangle = ps[i].\mathsf{decode}()}{ps[i].s.\mathsf{execute}(\langle \mathsf{Ld}, a, dst\rangle, m[a])}$$

SC-St rule (St execution).

$$\frac{\langle \mathsf{St}, a, v\rangle = ps[i].\mathsf{decode}()}{ps[i].s.\mathsf{execute}(\langle \mathsf{St}, a, v\rangle);\ m[a] \Leftarrow v}$$

Figure 3-2: SC operational semantics

3.4 TSO Model

Figure 3-3 shows the states and structure of TSO model. In addition to register state s, each processor now contains a store buffer sb. sb is an unbounded buffer of $\langle$address, value$\rangle$ pairs, each representing a pending store. The following methods are

defined on sb:

- empty(): returns True when sb is empty.
- exist(a): returns True if address a is present in sb.
- getYoungest(a): returns the store data of the youngest store to address a in sb.
- enq(a, v): enqueues the $\langle$address, value$\rangle$ pair $\langle a, v\rangle$ into sb.
- deq(): deletes the oldest store from sb, and returns its $\langle$address, value$\rangle$ pair.

Notice that the above deq() method not only updates the state of sb, but also returns a value. For this kind of *action-value* method, we use "$\leftarrow$" to assign its return value to a free variable. For example, $\langle a, v\rangle \leftarrow sb$.deq() assigns the return value of deq() to pair $\langle a, v\rangle$.

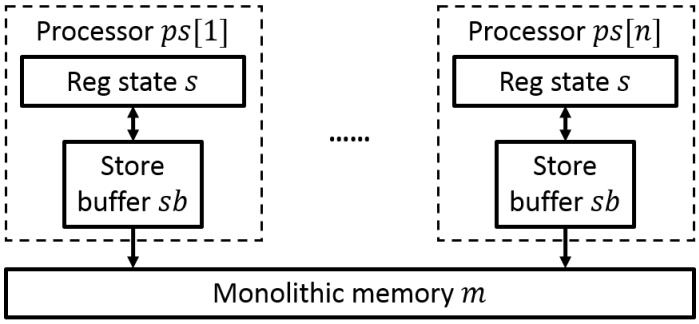

Figure 3-3: TSO model structure

In order to enforce instruction ordering in accessing the newly added store buffer, we extend our instruction set with the memory fence instruction called Commit which flushes the local store buffer.

Figure 3-4 shows the operational semantics of TSO. The first four rules are instantaneous execution of four types of decoded instructions, while the last rule handles the interaction between the store buffer and the monolithic memory.

According to the TSO-Ld rule, Ld a first tries to read the youngest store to address a in the local store buffer, and if sb does not contain a then it reads the monolithic memory m. TSO adds new behavior to SC because buffering stores in sb essentially

TSO-Nm rule (Nm execution).

$$\frac{\langle \mathsf{Nm}, op, regs\rangle = ps[i].\mathsf{decode}()}{ps[i].s.\mathsf{execute}(\langle \mathsf{Nm}, op, regs\rangle)}$$

TSO-Ld rule (Ld execution).

$$\frac{\begin{array}{rl} & \langle \mathsf{Ld}, a, dst\rangle = ps[i].\mathsf{decode}() \\ v = & \textbf{if } ps[i].sb.\mathsf{exist}(a) \textbf{ then } ps[i].sb.\mathsf{getYoungest}(a) \\ & \textbf{else } m[a] \end{array}}{ps[i].s.\mathsf{execute}(\langle \mathsf{Ld}, a, dst\rangle, v)}$$

TSO-St rule (St execution).

$$\frac{\langle \mathsf{St}, a, v\rangle = ps[i].\mathsf{decode}()}{ps[i].s.\mathsf{execute}(\langle \mathsf{St}, a, v\rangle);\ ps[i].sb.\mathsf{enq}(a, v)}$$

TSO-Com rule (Commit execution).

$$\frac{\langle \mathsf{Commit}\rangle = ps[i].\mathsf{decode}();\ \mathsf{when}(ps[i].sb.\mathsf{empty}())}{ps[i].s.\mathsf{execute}(\langle \mathsf{Commit}\rangle)}$$

TSO-DeqSb rule (dequeue store buffer).

$$\frac{\mathsf{when}(\neg ps[i].sb.\mathsf{empty}())}{\langle a, v\rangle \leftarrow ps[i].sb.\mathsf{deq}();\ m[a] \Leftarrow v}$$

Figure 3-4: TSO operational semantics

allows a load to be executed before an older store accesses the monolithic memory. Namely, TSO permits a load to overtake a store. The Commit instruction forces older stores to be flushed from store buffer into monolithic memory before any following instructions can execute. It should be noted that store atomicity [12] is broken at program level due to the store buffer even though all accesses on the monolithic memory are atomic.

Enable Store-Store reordering: We can introduce Store-Store reordering in TSO to form the related model known as Partial Store Order (PSO)[1]. Only the TSO-DeqSb rule should be changed such that the store buffer can commit the oldest store of any address, instead of the oldest one of all stores, to the monolithic memory. Specifically, we use the following two methods instead of deq() to delete entries from store buffer *sb*:

[1]The PSO model described here is somewhat different from SPARC PSO [46, 47] regarding fences.

- getAnyAddr(): returns any store address present in sb. If sb is empty, it returns ϵ.

- removeOldest(a): deletes the oldest store to address a from sb, and returns its store data.

Thus we can dequeue stores for different addresses from the same store buffer to monolithic memory out of order, namely reordering stores. The substitute of the TSO-DeqSb rule is the PSO-DeqSb rule shown in Figure 3-5

PSO-DeqSb rule (dequeue store buffer).

$$\frac{a = ps[i].sb.\mathsf{getAnyAddr}();\ \mathsf{when}(a \neq \epsilon)}{v \leftarrow ps[i].sb.\mathsf{removeOldest}(a);\ m[a] \Leftarrow v}$$

Figure 3-5: Rule for dequeuing store buffer in PSO

Comparing operational semantics and reordering axioms: I^{2}E operational semantics with monolithic memory simply and accurately define a memory model and one can prove all the reordering axioms induced by the operational semantics. The converse is not true, given a set of reordering axioms it is quite difficult to know if all the relevant aspects of the operational semantics have been captured. For example, the reordering axiom of TSO (i.e. loads overtaking stores) cannot capture the subtlety in TSO operational semantics that a load may bypass data from the older store to the same address even if it overtakes the store. A lot of difficulty in understanding weak memory models arises from their complex axiomatic descriptions.

Chapter 4

WMM Model

WMM, the weak memory model we are proposing, additionally allows Load-Load reordering compared to PSO. Despite the new reordering, we still want to describe WMM using I^{2}E and monolithic memory for simplicity and independence of vendors. When two loads appear to be reordered, the younger one should read a stale value when it is executed in the I^{2}E description. To model this behavior, we introduce a conceptual device called *invalidation buffer* to each processor, which holds stale values observable by the processor.

4.1 Structure of WMM

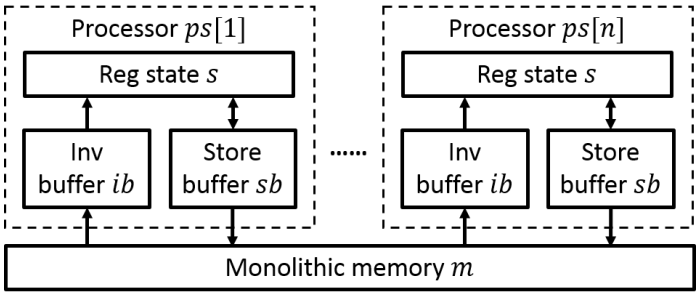

Figure 4-1: WMM model structure

The structure and states of WMM are shown in Figure 4-1. The change from PSO to WMM is to add an invalidation buffer ib to each processor. ib is an unbounded

buffer of $\langle$address, value$\rangle$ pairs, each representing a stale memory value for an address that can be observed by the processor. The following methods are defined on ib:

- insert(a, v): inserts $\langle$address, value$\rangle$ pair $\langle a, v\rangle$ into ib.
- getRand(a, m): returns either a random stale value for address a present in ib or monolithic memory value $m[a]$. The choice is arbitrary.
- clear(): removes all contents from ib to make it empty.
- removeAddr(a): removes all (stale) values for address a from ib.

The insert function is used to store stale values into ib, and the getRand function allows loads to access stale values and get reordered with older instructions. The clear and removeAddr functions are called when ordering needs to be enforced.

Similar to the introduction of Commit fences in TSO, we extend our instruction set with the memory fence instruction called Reconcile, which flushes the local invalidation buffer (i.e. invoke ib.clear()), to enforce ordering by preventing younger loads from accessing stale values.

4.2 Operational Semantics of WMM

Figure 4-2 shows the operational semantics of WMM. The first five rules describe the instantaneous execution of the decoded instructions, while the last one commits stores from store buffer to the monolithic memory.

According to the WMM-Ld rule, a load first attempts to read from the local store buffer in the same way as TSO-Ld rule does. If this attempt fails, the load reads either a random stale value for address a from the local invalidation buffer or the monolithic memory $m[a]$. The stale value from the invalidation buffer enables the load to be reordered with older instructions. The choice between invalidation buffer and monolithic memory is arbitrary in order to enable reordering of loads on the same address, which will be discussed in detail in Chapter 5.1. In the WMM-Rec rule, the Reconcile fence flushes the invalidation buffer.

WMM-Nm rule (Nm execution).

$$\frac{\langle \mathsf{Nm}, op, regs\rangle = ps[i].\mathsf{decode}()}{ps[i].s.\mathsf{execute}(\langle \mathsf{Nm}, op, regs\rangle)}$$

WMM-Ld rule (Ld execution).

$$\frac{\begin{array}{c}\langle \mathsf{Ld}, a, dst\rangle = ps[i].\mathsf{decode}() \\ v = \quad \textbf{if } ps[i].sb.\mathsf{exist}(a) \textbf{ then } ps[i].sb.\mathsf{getYoungest}(a) \\ \qquad \textbf{else } ps[i].ib.\mathsf{getRand}(a, m)\end{array}}{ps[i].s.\mathsf{execute}(\langle \mathsf{Ld}, a, dst\rangle, v)}$$

WMM-St rule (St execution).

$$\frac{\langle \mathsf{St}, a, v\rangle = ps[i].\mathsf{decode}()}{ps[i].s.\mathsf{execute}(\langle \mathsf{St}, a, v\rangle);\ ps[i].sb.\mathsf{enq}(a, v)}$$

WMM-Com rule (Commit execution).

$$\frac{\langle \mathsf{Commit}\rangle = ps[i].\mathsf{decode}();\ \mathsf{when}(ps[i].sb.\mathsf{empty}())}{ps[i].s.\mathsf{execute}(\langle \mathsf{Commit}\rangle)}$$

WMM-Rec rule (Reconcile execution).

$$\frac{\langle \mathsf{Reconcile}\rangle = ps[i].\mathsf{decode}()}{ps[i].ib.\mathsf{clear}();\ ps[i].s.\mathsf{execute}(\langle \mathsf{Reconcile}\rangle)}$$

WMM-DeqSb rule (dequeue store buffer).

$$\frac{a = ps[i].sb.\mathsf{getAnyAddr}();\ oldV = m[a];\ \mathsf{when}(a \neq \epsilon)}{\begin{array}{c}ps[i].ib.\mathsf{removeAddr}(a);\ v \leftarrow ps[i].sb.\mathsf{removeOldest}(a);\ m[a] \Leftarrow v \\ \forall j \neq i.\ ps[j].ib.\mathsf{insert}(a, oldV)\end{array}}$$

Figure 4-2: WMM operational semantics

The WMM-DeqSb rule removes the oldest store for any address from store buffer and commits it to monolithic memory. It also performs two other actions simultaneously: remove all stale values for address a from the local invalidation buffer and insert the original memory value in $m[a]$ into the invalidation buffers of all other processors. Removing the stale values for a from the local invalidation buffer is essential to maintain the correctness of the single-threaded execution. Insertion of the stale $m[a]$ value in other invalidation buffers allows Ld a in other processors to effectively get reordered with older instructions.

4.3 Reordering Axioms of WMM

WMM executes instructions instantaneously and in order. However, because of store buffers (sb) and invalidation buffers (ib), processor j can see the effect of loads and stores on some processor i ($i \neq j$) in a different order than the program order on processor i. We explain the reorderings permitted by the definition of WMM using examples.

Example: mutual exclusion. Figure 4-3 shows the kernel of Dekker's algorithm, which guarantees mutual exclusion by ensuring registers r_1 and r_2 cannot both be zero after the program finishes. If no reordering were allowed, this invariant would obviously hold. Now consider the following scenario in WMM, in which both stores (I_1 and I_3) are executed but stay in store buffers (not committed to memory) and then following loads (I_2 and I_4) are executed. In this case, the memory values of both a and b would be zero. It is as if the load overtook the older store on both processors.

We can force the stores to be performed before the following loads by inserting Commit fences (I_5 and I_7 in Figure 4-4). However, the stale values of a and b will show up in the invalidation buffers, and can be read by subsequent loads. Basically, the program will behave as if the load overtook the preceding Commit fence and store on both processors.

If we want to prevent this reordering, we also need to insert Reconcile fences (I_6 and I_8 in Figure 4-4). Once these fences are inserted, the invariant will be restored. This implies that a load cannot overtake a Reconcile fence.

Example: message passing. A popular paradigm in programming is to signal an event by writing a shared variable as shown in Figure 4-5. P1 writes data 100 to addresses a, and then signals P2 that the data has been written by setting a flag at address f to 1. P2 waits for the value of f to change and then reads the data. Due to the store buffer in P1, the flag may be written to the monolithic memory before the data is written, and P2 will not see the new data. It is as if the two stores on P1 are reordered. A Commit after data write (I_6 in Figure 4-6) will force the data to be committed to memory, namely forbidding the reordering of stores on P1. However,

Initial: $m[a] = 0,\ m[b] = 0$	
Processor P1	Processor P2
I_1 : St a 1	I_3 : St b 1
$I_2 : r_1 =$ Ld b	$I_4 : r_2 =$ Ld a
Forbidden: $r_1 = 0 \ \wedge \ r_2 = 0$	

Figure 4-3: Dekker's algorithm in SC

Initial: $m[a] = 0,\ m[b] = 0$	
Processor P1	Processor P2
I_1 : St a 1	I_3 : St b 1
I_5 : Commit	I_7 : Commit
I_6 : Reconcile	I_8 : Reconcile
$I_2 : r_1 =$ Ld b	$I_4 : r_2 =$ Ld a
Forbidden: $r_1 = 0 \ \wedge \ r_2 = 0$	

Figure 4-4: Dekker's algorithm in WMM

Initial: $m[a] = 0,\ m[f] = 0$	
Processor P1	Processor P2
I_1 : St a 100	$I_3 : r_1 =$ Ld f
I_2 : St f 1	I_4 : if $(r_1 \neq 1)$ goto I_3
	$I_5 : r_2 =$ Ld a
Forbidden: $r_2 = 0$	

Figure 4-5: Message passing in SC

Initial: $m[a] = 0,\ m[f] = 0$	
Processor P1	Processor P2
I_1 : St a 100	$I_3 : r_1 =$ Ld f
I_6 : Commit	I_4 : if $(r_1 \neq 1)$ goto I_3
I_2 : St f 1	I_7 : Reconcile
	$I_5 : r_2 =$ Ld a
Forbidden: $r_2 = 0$	

Figure 4-6: Message passing in WMM

Initial: $m[a] = 0,\ m[b] = 0$	
Processor P1	Processor P2
$I_1 : r_1 =$ Ld b	$I_3 : r_2 =$ Ld a
I_2 : St a 1	I_4 : St b 1
Forbidden: $r_1 = 1 \ \wedge \ r_2 = 1$	

Figure 4-7: Load buffering in WMM

P2 may still see stale data in the invalidation buffer, and the program behaves as if the two loads on P2 are reordered. To make sure that P2 reads data from memory, it must execute a Reconcile (I_7 in Figure 4-6) before reading the data.

Example: load buffering. Figure 4-7 shows the load buffering example, which is the dual of Dekker's algorithm. In WMM, P1 will execute I_1 before I_2 writes the monolithic memory, and P2 will execute I_3 before I_4 writes the monolithic memory. Therefore, it is impossible for both I_1 and I_3 to read from stores. This implies that WMM prohibits the reordering of stores overtaking loads. In fact, such reordering is not expressible in I^2E models, because by definition we cannot see the effects of stores yet to be executed.

4.3.1 Summarizing Reordering Axioms

By studying above examples, we can summarize the following axioms on instruction reordering in WMM:

1. Loads can overtake loads, stores and Commit fences, but cannot overtake any Reconcile fence.

2. Stores can only overtake stores.

The second axiom is easy to see. To better illustrate the first axiom, we can consider simple programs without control or data dependency. In case of such programs, the following lemma shows the possible reorderings of loads in WMM.

Lemma 1. *For any program P, which only contains fences (i.e.* Commit *and* Reconcile*) and memory access instructions (i.e.* Ld *and* St*) with fixed load/store addresses and store data, reordering a* Ld *instruction with its direct predecessor, which is neither a* Reconcile *fence nor a* St *instruction to the same address, will not generate any new result in the WMM model.*

Proof. Assume P contains two consecutive instructions I_1 and I_2 on processor $ps[i]$, and I_2 follows I_1. Specifically, I_2 is a Ld to address a, while I_1 is neither a Reconcile fence nor a St to address a. After reordering I_1 and I_2, we can get a new program P'. Our goal is to prove that any result R of P' is also a result of P in WMM. Here "result" refers to the return value of each load as well as the final memory states.

First of all, there must be a sequence of rules E' to run program P' and get result R. The guideline is to construct another sequence of rules E, which can run program P to get result R, based on E'. We assume that r_2 is the WMM-Ld rule in E' to execute I_2, and r_1 is the rule in E' to execute I_1. We further assume x is the load result of I_2 in E'.

The construction of E is simply to reschedule r_2 to fire right after r_1 in E'. It is obvious that the behaviors of all rules other than r_2 in E are the same as those in E'. Thus we only need to consider whether r_2 can still get data x in E. We can perform a case analysis on where r_2 gets data x in E'.

We first consider the case that r_2 gets it from store buffer $ps[i].sb$ in E'. Since the store forwarding x to I_2 in E' must be the youngest store to address a in $ps[i].sb$ and I_1 is not a Reconcile fence, this value x must be in store buffer $ps[i].sb$ or monolithic memory $m[a]$ or invalidation buffer $ps[i].ib$ after r_1 fires in E. Therefore r_2 in E can still get value x.

As for other cases where r_2 reads x from monolithic memory or $ps[i].ib$ in E', we can use similar arguments to show that r_2 in E still gets data x. □

It is interesting to note that the operational semantics of WMM capture the effects of instruction reordering without ever requiring instructions to be executed out-of-order. Later on we will show how WMM can be implemented using standard reorder buffers and store buffers without any need for the invalidation buffer. In fact, invalidation buffer is simply an abstraction to model out-of-order execution.

4.4 Synchronization Instructions

Processors often provide special instructions to implement synchronization efficiently. The most common instructions fall into two categories: atomic read-modify-write and load-linked/store-conditional (LL/SC) pair. Here we only show how to extend WMM to include read-modify-write instructions. The LL/SC pair can be incorporated in a similar way.

A read-modify-write instruction $\langle \mathsf{RMW}, a, f, x, dst \rangle$ may be described as follows. It first reads address a to get value v, then writes $f(v, x)$ back into $m[a]$, and finally updates the dst register using v. For example, the XCHG instruction in x86 exchanges register and memory values, which corresponds to function $f(v, x)$ returning x (i.e. the original register value). And the LOCK XADD instruction in x86 fetches and adds to a memory location, which corresponds to $f(v, x)$ returning $v + x$.

The additional rule to execute RMW is shown in Figure 4-8. We need to recognize that memory read and write must be performed in the monolithic memory, so address a cannot be in the store buffer. Similar to the WMM-DeqSb rule, we removes all stale

values of address a from the local invalidation buffer, and insert the stale $m[a]$ into the invalidation buffers of all other processors

WMM-RMW rule (RMW execution).

$$\frac{\langle \mathsf{RMW}, a, f, x, dst\rangle = ps[i].s.\mathsf{decode}();\ v = m[a] \quad \mathsf{when}(\neg ps[i].sb.\mathsf{exist}(a))}{ps[i].s.\mathsf{execute}(\langle \mathsf{RMW}, a, f, x, dst\rangle, v);\ m[a] \Leftarrow f(v,x) \quad ps[i].ib.\mathsf{removeAddr}(a);\ \forall j \neq i.\ ps[j].ib.\mathsf{insert}(a, v)}$$

Figure 4-8: Rule for execution of RMW in WMM

4.5 Well-synchronized Programs

In well-synchronized programs, the critical section is protected by locks. In order to run such programs in WMM, we only need to add a Reconcile fence after acquiring the lock and a Commit fence before releasing the lock. The general procedure is:

acquire lock → Reconcile fence → critical section → Commit fence → release lock

Figure 4-9 shows an implementation of LOCK() and UNLOCK() functions for a spin lock in WMM. Here input argument a is the memory address of the lock variable. If the lock variable is equal to 0, then the lock is free. Otherwise the lock is held by some thread.

LOCK(a)	
I_1:	$r_1 = 1$
I_2:	XCHG a r_1
I_3:	if ($r_1 \neq 0$) goto I_2
I_4:	Reconcile

UNLOCK(a)	
I_5:	Commit
I_6:	St a 0

Figure 4-9: Spin lock implementation for WMM

In the LOCK() function, I_2 and I_3 are spinning to acquire the lock. After getting the lock, the Reconcile fence at I_4 prevents loads in the following critical section from reading stale memory values which have been overwritten before lock acquirement.

In the UNLOCK() function, the store in I_6 just releases the lock. The Commit fence at I_5 ensures that all stores in the above critical section have been committed to monolithic memory before I_6 releases the lock.

Chapter 5

WMM Processor Implementation

In the definition of WMM, we have used the invalidation buffer as a convenient way of modelling out-of-order execution, and have used monolithic memory as an abstraction of real memory system. In spite of these conceptual devices in the definition, WMM implementation can be built with the conventional reorder buffer (ROB), store buffer and coherent memory hierarchy. We will show that the implementation will not step beyond WMM even with the most aggressive optimizations such as load-value prediction.

5.1 Reorder Buffer (ROB)

If we are not concerned with fences, the WMM multi-processor can directly use the ROB of the most aggressive uniprocessor implementation. In other words, WMM implementation can issue a load as soon as the load address is known, even if the address is a predicted value. Moreover, it does not need any additional logic to keep track of the accessed memory location after the load returns, such as monitoring L1 cache eviction. Due to the in-order execution property of WMM, we only consider ROBs that commit instructions in order. We refer to the slot containing the oldest instruction in ROB as the *commit slot*.

The major operations of an ROB are committing stores to the store buffer, and executing loads. Both operations can be related to the rules in WMM. The commit

of a store corresponds to the WMM-St rule. The relation between a load execution and the WMM-Ld rule depends on how the load in ROB gets its value:

- If the load in ROB bypasses from a store, which has not written the memory when the load commits, the load execution corresponds to bypassing from store buffer in the WMM-Ld rule.
- If the load in ROB bypasses from a store, which has written the memory before the load commits and the memory location is never overwritten afterwards until the load commits, the load execution corresponds to reading monolithic memory in the WMM-Ld rule.
- If the load in ROB reads from memory and the memory location is never overwritten afterwards until the load commits, the load execution also corresponds to reading monolithic memory in the WMM-Ld rule.
- In all other cases, the load reads a stale value as compared to the memory value when the load commits, so the execution corresponds to accessing the invalidation buffer in the WMM-Ld rule.

The above relation shows that WMM implementation can use a uniprocessor ROB. To better illustrate the resilience of WMM, we study three subtle non-SC behaviors induced by the microarchitectural optimizations in ROB, and we will show all these behaviors are allowed by WMM.

Memory dependency speculation: Figure 5-1 shows a non-SC behavior induced by memory dependency speculation. In implementation, I_1 must have written memory before I_3 is inserted to the store buffer, because I_2 reads the store data of I_1. If P3 stalls I_6 until I_5 resolves its store address to avoid potential memory dependency violation, I_6 can never return the stale value 0 of a. However, with memory dependency speculation, P3 can issue I_6 to memory without knowing whether the store address of I_5 is the same as the load address of I_6. Thus I_6 can get the stale value 0 of a even before I_4 is issued. This behavior is allowed by WMM, because I_6 can access the stale value 0 of a from the invalidation buffer.

Load-value prediction: Figure 5-2 shows another non-SC behavior which is the result of load-value prediction. If there were no load-value prediction in implementation, the load address of I_5 would remain unknown until I_4 returned from memory, so I_5 would have to get the up-to-date value 1 of a. However, with load-value prediction, P3 can predict the load result of I_4 to be 1 so as to issue I_5 to memory even before I_4 is issued, and I_5 will get the stale value 0 of a. After I_4 finally returns from memory, P3 verifies the correctness of the load-value prediction on I_4, and the result of I_5 can be kept. This behavior is also allowed by WMM, because I_5 can access the stale value 0 of a from the invalidation buffer.

Reordering loads to the same address: In uniprocessor, a load only needs to observe the dependency on earlier stores to the same address, so the ROB can issue loads for the same address out of order. This leads to the non-SC behavior shown in Figure 5-3, where P2 issues I_3 before I_2.

This behavior is still allowed by WMM since the WMM-Ld rule can make arbitrary choice between the monolithic memory and invalidation buffer. Specifically, I_1 first modifies monolithic memory and inserts the stale value into P2's invalidation buffer. Then I_2 reads the up-to-date value from monolithic memory without flushing the invalidation buffer, and finally I_3 reads the stale value from the invalidation buffer.

Fence implementation: Fences in WMM can also be easily implemented in the ROB design. The Commit fence is the same as a NOP instruction except that it cannot be committed from ROB until all older stores on this processor have been committed to the coherent memory. The Reconcile fence is also similar to a NOP instruction. The difference is that the Reconcile fence will stall all younger loads, which cannot bypass from stores younger than the fence, from execution. Neither the Commit nor Reconcile fence gets into the store buffer or coherent memory.

5.2 Store Buffer

The store buffer holds stores, which have been committed by the ROB but are not yet committed to the coherent memory. Since WMM allows the reordering of stores, the

Initial: $m[a] = 0,\ m[b] = 0,\ m[c] = 0$		
Processor P1	Processor P2	Processor P3
$I_1 : \mathsf{St}\ a\ 1$	$I_2 : r_1 = \mathsf{Ld}\ a$ $I_3 : \mathsf{St}\ b\ 1$	$I_4 : r_2 = \mathsf{Ld}\ b$ $I_5 : \mathsf{St}\ (r_2 + c - 1)\ 1$ $I_6 : r_3 = \mathsf{Ld}\ a$
Allowed: $r_1 = 1\ \wedge\ r_2 = 1\ \wedge\ r_3 = 0$		

Figure 5-1: Behavior by memory dependency speculation

Initial: $m[a] = 0,\ m[b] = 0$		
Processor P1	Processor P2	Processor P3
$I_1 : \mathsf{St}\ a\ 1$	$I_2 : r_1 = \mathsf{Ld}\ a$ $I_3 : \mathsf{St}\ b\ 1$	$I_4 : r_2 = \mathsf{Ld}\ b$ $I_5 : r_3 = \mathsf{Ld}\ (r_2 + a - 1)$
Allowed: $r_1 = 1\ \wedge\ r_2 = 1\ \wedge\ r_3 = 0$		

Figure 5-2: Behavior by load-value prediction

Initial: $m[a] = 0$	
Processor P1	Processor P2
$I_1 : \mathsf{St}\ a\ 1$	$I_2 : r_1 = \mathsf{Ld}\ a$ $I_3 : r_2 = \mathsf{Ld}\ a$
Allowed: $r_1 = 1\ \wedge\ r_2 = 0$	

Figure 5-3: Behavior by reordering loads to the same address

store buffer in WMM implementation can merge stores for the same cache line, and can commit stores for different addresses to memory out of order. The interaction between the store buffer and the L1 cache is quite subtle, and we leave the discussion to Chapter 6.

It should be noted that the store buffer in the WMM model is private to each logical processor. A multithreaded WMM processor will violate the model if every entry of the store buffer can be accessed by all threads sharing the physical core. To avoid such violation, we need to tag thread IDs to stores in the shared store buffer to prevent a thread from accessing stores of any other threads.

Chapter 6

Separating Memory Model from Cache Coherence

The interaction between processors and memory system varies across implementations, which complicates the reasoning about whether an implementation conforms to a certain memory model. To simplify the reasoning, we will re-draw the boundary between processors and memory so that the memory is only responsible for *coherence* while processors are fully responsible for ordering memory accesses to ensure compliance with the memory model. We refer to the memory system after the new partition as *purely coherent memory* (PCM) and show how existing implementations can be logically partitioned to take advantage of PCM for reasoning about memory model issues.

6.1 Specification of PCM

A high-performance memory system is always pipelined, i.e. handles many requests concurrently, while our monolithic memory in I^2E definitions does not. Furthermore, the order in which a memory system processes requests is either not visible to or cannot be relied upon by the processors. We define the semantics of PCM operationally using the monolithic memory. Intuitively the idea can be understood in terms of a monolithic memory m and a *memory request buffer* ($mrb[i]$, $i = 1 \ldots n$) for each port.

The monolithic memory arbitrarily selects *any* request from *any* mrb and processes it instantaneously. Since the memory can hold many requests, each request has to be assigned a unique tag so that responses can be attached to requests unambiguously.

For correct functioning, each processor has to exercise control over which requests it enters into its mrb, since PCM can process requests out of order. For example, no processor should enter the second store for the same address until it gets a response back for the first store. Whether a processor can enter a store for a different address before getting the response for the previous store depends upon the memory model the processor wants to observe. Similar reasoning is needed to issue load requests to PCM.

We now formalize the definition of PCM. Figure 6-1 shows that PCM has one port connected to each processor. Processor i sends requests to port i using the following methods, where t are the unique tags of processor requests:

- $\mathsf{reqLd}(a, t)$: inserts load request $\langle \mathsf{Ld}, a, t \rangle$ into $mrb[i]$, where a is the load address.
- $\mathsf{reqSt}(a, v, t)$: inserts store request $\langle \mathsf{St}, a, v, t \rangle$ into $mrb[i]$, where a is the store address and v is the store data.

The PCM sends responses back by calling the following methods of the processor, where t are the tags of original requests:

- $\mathsf{respLd}(res, t)$: res is the result of the load request.
- $\mathsf{respSt}(a, t)$: a is the store address of the store request.

When the respLd and respSt methods are invoked, the processor will take action (e.g. satisfying a load in ROB or removing a store from store buffer) according to the response. The operational semantics of PCM is shown in Figure 6-2, where getRand() returns any entry in $mrb[i]$.

The order in which the rules of PCM fire imposes a total order on all memory accesses, which is what we mean by *coherence*. And this order will correspond to the order of accesses to the monolithic memory used in the I^{2}E definitions of SC, TSO and WMM.

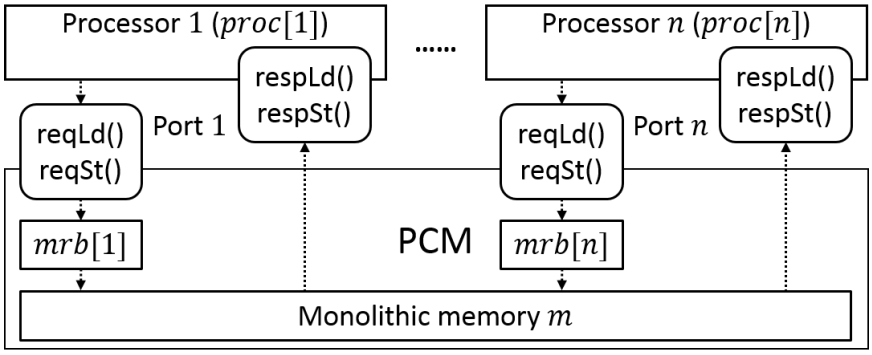

Figure 6-1: PCM interface and structure

PCM-Ld rule (load request processing).

$$\frac{\langle \mathsf{Ld}, a, t\rangle = mrb[i].\mathsf{getRand}()}{mrb[i].\mathsf{remove}(\langle \mathsf{Ld}, a, t\rangle);\ proc[i].\mathsf{respLd}(m[a], t)}$$

PCM-St rule (store request processing).

$$\frac{\langle \mathsf{St}, a, v, t\rangle = mrb[i].\mathsf{getRand}()}{mrb[i].\mathsf{remove}(\langle \mathsf{St}, a, v, t\rangle);\ m[a] \Leftarrow v;\ proc[i].\mathsf{respSt}(a, t)}$$

Figure 6-2: PCM operational semantics

Atomicity violations because of delayed responses: The store buffer of a WMM processor needs the store response from PCM to remove the completed store from it. The removal of a store from the store buffer and the updating of memory must appear to happen atomically. To understand what happens when this atomicity is violated, consider the case when another processor updates the same address between the time store response is generated and the time store buffer entry is deleted. Then the store buffer will contain a stale value for that address, violating WMM semantics.

To ensure the atomicity, we can typically restrict the delay of propagating the response to the store buffer, because the store is performed in L1, which is close to the store buffer, for a common write-back cache hierarchy. However, if the delay can be arbitrarily long, a processor must conservatively assume that any store already issued to PCM may have become stale even before the response comes back. It deals with this possibility by making such stores inaccessible to younger loads. Namely, if a load attempts to bypass from a store in the store buffer, the load will be stalled from execution if the store has been issued to the PCM.

6.2 Mapping Real Memory Systems to PCM

The behavior of a real memory system may not match the specification of PCM exactly, but we can transform it to a valid PCM by conceptually moving part of the memory system to the processor side. Here we study two examples on transforming real memory systems to PCM.

Relying on the order of processing memory requests: In a real implementation, a processor may rely on ordered processing of requests by the memory system, which will disqualify such a memory system to be a PCM. To understand how a system does it, consider a memory system composed of write-back caches. Processing of a store has two distinct phases: (1) get exclusive permission for the cache line, and (2) update the cache line. It should be clear that phase (1) can be done for multiple distinct addresses in any order. Ordering of stores can be enforced by ordering phase (2) operations starting from L1. If L1 updates memory in order, so will higher-level caches. A TSO implementation can rely on its L1 cache to process store requests in order and forward data from in-flight store requests to subsequent load requests. By exploiting this ordering, a processor can dequeue the oldest store from the store buffer as soon as the store is issued to L1, as opposed to waiting for the response to come back from the memory system.

Such a memory system can be trivially transformed into a PCM by conceptually treating the MSHR (miss status handling register) entries that hold in-flight store requests as part of the processor's store buffer.

Write-through L1 cache: A memory system with write-through L1 caches is not a PCM since store atomicity is broken inside it. When the write-through policy is used, the store request is always sent to the L2, and it also directly writes L1 if the cache line is valid. The store is removed from the store buffer when L2 finishes processing it. Store atomicity is broken, because the modification on L1 made by this store can be observed by subsequent loads issued by any processor sharing the same L1 while being invisible to other processors.

Such a memory system can be transformed into PCM by considering the "dirty"

data in L1 as part of the processor's store buffer. Then the remaining part of the memory system, including the "clean" data in L1, forms a valid PCM.

Chapter 7

Comparison with other Weak Memory Models

7.1 Comparison with Release Consistency

Release Consistency (RC) [23] is defined in terms of the ordering of when memory accesses are *performed with respect to* certain processors. We call such type of definition *event ordering* (EO) as opposed to I^2E. RC enforces ordering by tagging memory accesses as *acquire* or *release* operations, which are collectively referred to as *special accesses*.

The definition of being performed is subtle in EO because the memory abstraction is not monolithic. Namely, a store may become visible to different processors (which do not issue the store) at different time. The exact definitions of "*performed with respect to*", "*performed*", and "*globally performed*", and the definitions of SC and RC using EO can be found in [23]. RC comes with two versions: RC_{pc} where special accesses are processor consistent, and RC_{sc} where special accesses are sequentially consistent. Here we only discuss RC_{sc} for simplicity.

RC is weaker than WMM, because RC allows a store to overtake a load and allows store access to be non-atomic inside the memory, while WMM forbids both behaviors.

7.1.1 Comparing understandability

WMM is more understandable than RC when being used to reason about racy programs, because the non-monolithic memory in RC breaks the atomicity of memory accesses. One can view WMM as a tradeoff between model weakness and understandability. To illustrate this point, we study how to insert fences in WMM and how to tag accesses in RC to enforce SC behavior for the independent-read-independent-write (IRIW) example shown in Figure 7-1. The behavior in Figure 7-1 is disallowed under SC because P3 and P4 must observe the same order of the memory writes by I_1 and I_2.

The forbidden behavior under SC is allowed in WMM because both I_4 and I_6 can read stale values from invalidation buffers. To prohibit this behavior, we simply need to insert Reconcile fences I_7 and I_8 as shown in Figure 7-2. To understand the reason, let's assume I_1 writes the monolithic memory before I_2 does. Since I_5 reads from I_2, the stale value 0 of a must have been inserted into the invalidation buffer of P4 when I_5 is executed. Then I_8 will flush this stale value from the invalidation buffer, so I_6 can never read it. A similar argument applies to the situation where I_2 writes the monolithic memory before I_1 does. As we can see, the reasoning using $\mathrm{I}^2\mathrm{E}$ and monolithic memory is very simple.

RC also allows the forbidden result in SC by permitting I_4 and I_6 to be performed earlier than any other instruction. In contrast to the simplicity in WMM, tagging accesses in RC to enforce SC behavior is more convoluted. Figure 7-3 shows a wrong tagging scheme, where I_3 and I_5 are tagged as acquire operations. The intention is to force P3 and P4 to perform loads in program order, so they are not expected to observe different orders of I_1 and I_2. However, the non-monolithic memory in RC makes it possible that I_1 is not performed with respect to P4 when I_6 is performed, and that I_2 is not performed with respect to P3 when I_4 is performed. Therefore the non-SC behavior is still allowed. This pitfall is the direct result of non-monolithic memory abstraction.

A correct tagging scheme is given in Figure 7-4, where I_4 and I_6 are also tagged as

Initial: $m[a] = 0,\ m[b] = 0$			
Processor P1	Processor P2	Processor P3	Processor P4
I_1 : St a 1	I_2 : St b 1	$I_3 : r_1 =$ Ld a	$I_5 : r_3 =$ Ld b
		$I_4 : r_2 =$ Ld b	$I_6 : r_4 =$ Ld a
Forbidden: $r_1 = 1 \ \wedge \ r_2 = 0 \ \wedge \ r_3 = 1 \ \wedge \ r_4 = 0$			

Figure 7-1: IRIW in SC

Initial: $m[a] = 0,\ m[b] = 0$			
Processor P1	Processor P2	Processor P3	Processor P4
I_1 : St a 1	I_2 : St b 1	$I_3 : r_1 =$ Ld a	$I_5 : r_3 =$ Ld b
		I_7 : Reconcile	I_8 : Reconcile
		$I_4 : r_2 =$ Ld b	$I_6 : r_4 =$ Ld a
Forbidden: $r_1 = 1 \ \wedge \ r_2 = 0 \ \wedge \ r_3 = 1 \ \wedge \ r_4 = 0$			

Figure 7-2: IRIW in WMM

Initial: $m[a] = 0,\ m[b] = 0$			
Processor P1	Processor P2	Processor P3	Processor P4
I_1 : St a 1	I_2 : St b 1	$I_3 : r_1 =$ Ld-acq a	$I_5 : r_3 =$ Ld-acq b
		$I_4 : r_2 =$ Ld b	$I_6 : r_4 =$ Ld a
Allowed: $r_1 = 1 \ \wedge \ r_2 = 0 \ \wedge \ r_3 = 1 \ \wedge \ r_4 = 0$			

Figure 7-3: IRIW in RC: wrong tagging

Initial: $m[a] = 0,\ m[b] = 0$			
Processor P1	Processor P2	Processor P3	Processor P4
I_1 : St a 1	I_2 : St b 1	$I_3 : r_1 =$ Ld-acq a	$I_5 : r_3 =$ Ld-acq b
		$I_4 : r_2 =$ Ld-acq b	$I_6 : r_4 =$ Ld-acq a
Forbidden: $r_1 = 1 \ \wedge \ r_2 = 0 \ \wedge \ r_3 = 1 \ \wedge \ r_4 = 0$			

Figure 7-4: IRIW in RC: correct tagging

acquire operations. One may wonder why adding these tags could solve the problem, since we have already enforced the program order on P3 and P4 in Figure 7-3. The subtlety is that both I_3 and I_4 are special accesses now, so they are sequentially consistent. According to the definition of SC in EO, I_3 must be *globally performed*, not just *performed*, before I_4 is performed. Thus I_1 should be performed with respect to all processors before I_4 is performed. Similarly, I_2 must be performed with respect to all processors before I_6 is performed. In this way, at least one of I_4 and I_6 will observe the up-to-date value 1, so the non-SC behavior is prohibited. As we can see, if one is not extremely familiar with the definition of RC, he/she may even fail to justify the correct tagging scheme.

Through the above reasoning process, we found WMM is much more understandable than RC. The primary drawback of RC in terms of understandability is the non-monolithic memory abstraction, which forces the definition to describe the reordering of events with respect to each processor.

7.1.2 Comparing implementation

Although RC is weaker than WMM, it is unclear whether the additional behaviors (i.e. non-monolithic memory and store overtaking load) allowed by RC could benefit implementations. As for the non-monolithic memory, perhaps it can only relax implementation in the case of multithreaded cores. This is because the stores in the store buffer are visible to all threads sharing the core but are invisible to other threads, exhibiting the non-monolithic memory behavior. However, WMM implementations can still use multithreaded cores with very little overhead, as described in Section 5.2. As for the store-overtaking-load behavior, it is disallowed by Intel's x86 memory model, and it cannot be observed on most Power processors [42] even though the memory model allows it. Thus the advantage of including these behaviors into the memory model is not obvious.

7.1.3 Summary

Although RC is weaker than WMM, WMM is much simpler than RC and the additional behavior allowed by RC may not benefit implementation.

7.2 Comparison with Power

Power is incomparable with WMM in terms of which model is weaker. On the one hand, Power is stronger than WMM in issuing loads for two reasons. First, Power cannot reorder dependent loads [28], i.e. it disallows the behavior in Figure 5-2. If a Power implementation employs load-value prediction to issue load I_5 in Figure 5-2 with the load address as a predicted value, it must check whether the load result of

I_5 becomes stale when the prediction of I_4 is verified. Second, Power enforces SC for memory accesses to the same address, so it prohibits the behavior in Figure 5-3 [42]. If a Power implementation issues a load and then resolves the load address of an older load to be the same as the just issued one, it must make sure that two loads appear to be performed sequentially. In contrast, WMM allows both behaviors and implementations do not need any additional logic in these cases.

On the other hand, Power is similar to RC and weaker than WMM in processing stores. Specifically, Power also employs a non-monolithic memory abstraction and allows stores to overtake loads. As discussed earlier, these two features may not benefit implementation, while the non-monolithic memory abstraction definitely complicates the model.

In summary, WMM is more understandable than Power, and has more area-efficient implementation than Power does.

Chapter 8

Comparing Performance with Strong Memory Models

It should be noted that any optimization permitted by a weak memory model can also be employed by a strong memory model implementation via speculation. For example, even though SC model dictates instructions to be executed in order, SC implementation can still speculatively issue a load from the middle of ROB, as long as the implementation checks whether the load result becomes stale at commit time and rolls back on mispeculation. Therefore the performance gap between the implementations of a weak memory model and a strong memory model is mostly determined by the frequency of speculation failure in the strong memory model implementation, which is highly dependent on benchmarks. As a preliminary evaluation, we only compare a moderate WMM implementation against well-known speculative implementations of SC and TSO using well-synchronized programs.

8.1 Evaluation Methodology

We evaluate the performance of SC, TSO and WMM implementations by running SPLASH-2x benchmarks [49, 1, 2] on an 8-core multiprocessor using ESESC simulator [10]. We ran only 12 out of 14 benchmarks because we could not compile ocean_ncp and volrend. We used sim-medium size inputs except for cholesky, fft and radix,

where we use sim-large size inputs. We run all benchmarks to completion without sampling.

The overall system configuration of the 8-core multiprocessor is shown in Table 8.1, and the configuration for each out-of-order core is shown in Table 8.2. All three models share these configurations; the difference between the implementations lies in how we implement loads and stores.

Cores	8 cores (@2GHz) with private L1 and L2 caches
L3 cache	4MB shared, MESI coherence, 64 byte cache line 8 banks, 16-way set-associative, LRU replacement 3-cycle tag, 10-cycle data (both pipelined) 5 cycles between cache bank and core (pipelined) Max 32 upgrade requests per bank
Memory	120-cycle latency, max 24 requests

Table 8.1: Multiprocessor system configuration

Front end	fetch + decode + rename 7-cycle pipelined latency in all 2-way superscalar, hybrid branch predictor
ROB	128 entries, 2-way issue/commit
Function units	2 ALUs, 1 FPU, 1 branch unit 1 load unit, 1 store unit 32-entry reservation station per unit
load queue	Max 32 loads
store buffer	Max 24 stores
Store set	8192-entry store set ID table (SSIT) 256-entry last fetched store table (LFST)
L1 D cache	32KB private, 1 bank, 64 byte cache line 4-way set associative, LRU replacement 1-cycle tag, 2-cycle data (pipelined) Max 32 upgrade and 8 downgrade requests
L2 cache	128KB private, 1 bank, 64 byte cache line 8-way set associative, LRU replacement 2-cycle tag, 6-cycle data (both pipelined) Max 32 upgrade and 8 downgrade requests

Table 8.2: Core configuration

Store implementation: We issue a store into the store buffer at the same time when issuing it to ROB, but only the stores committed by ROB can write memory. This avoids associative searches on ROB. The store buffer is a FIFO in SC and TSO while the store buffer in WMM employs the optimizations described in Section 5.2.

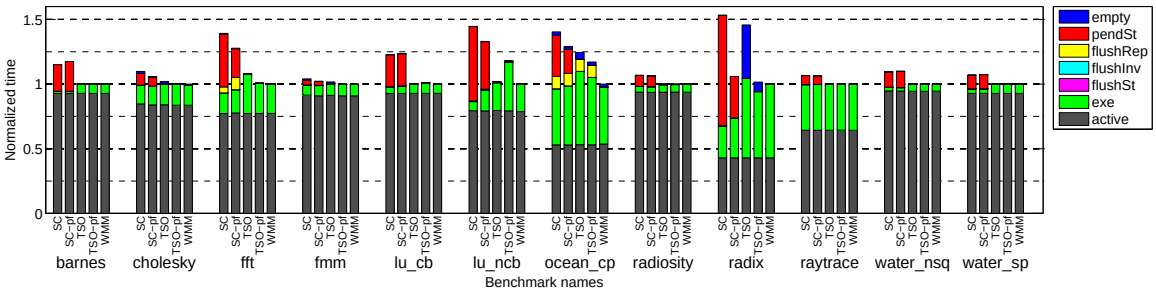

Figure 8-1: Normalized execution time and its breakdown at the commit slot of ROB

We also implement store prefetch as an optional feature for SC and TSO. We initiate a prefetch when a store resolves its address and the store buffer does not contain any other store on the same cache line. We use **SC-pf** and **TSO-pf** to denote implementations of SC and TSO with store prefetch.

Load implementation: All models execute loads speculatively, and use *store sets* [18] to predict memory dependency. For SC, a load cannot be committed when the store buffer contains any older store. For both SC and TSO, we monitor L1 cache eviction to kill speculative loads that violate the consistency model.

8.2 Evaluation Results

A common way to study the performance of memory models is to monitor the commit of instructions at the commit slot of ROB. Here are some reasons why an instruction may not commit in a given cycle:

- **empty**: The ROB is empty.
- **exe**: The instruction at the commit slot of ROB is still executing.
- **pendSt**: The Ld instruction (in SC) or Commit fence (in TSO or WMM) at the commit slot of ROB cannot commit due to pending older stores.
- **flushSt**: ROB is being flushed due to a memory dependency violation.
- **flushInv**: ROB is being flushed after cache invalidation caused by a remote store (only in SC or TSO).

- **flushRep**: ROB is being flushed after cache replacement (only in SC or TSO).

Figure 8-1 shows the execution time normalized to WMM and its breakdown at the commit slot of ROB. The total height of each bar represents the normalized execution time, and stacks represent different types of stall times added to the active committing time at the commit slot.

8.3 Performance: WMM *versus* SC

Figure 8-1 shows that WMM is much faster than both SC and SC-pf for most benchmarks. WMM can reach up to 1.53× performance of SC, with a geometric mean over all benchmarks of 1.20×. Store prefetch helps but WMM can still reach up to 1.33× performance of SC-pf, with a geometric mean of 1.14×. The reason why WMM outperforms SC is because a pending older store can block SC from committing loads. As shown in Figure 8-1, "pendSt" stall dominates in both SC and SC-pf.

8.4 Performance: WMM *versus* TSO

Figure 8-1 shows that in terms of geometric mean over all benchmarks WMM gives negligible improvement over TSO and TSO-pf. However, WMM never does worse than TSO or TSO-pf, and in some cases it shows up to 1.46× speedup over TSO and 1.18× over TSO-pf. In the following, we analyze the three benchmarks: lu_ncb, ocean_cp and radix, where WMM does better.

ocean_cp: In ocean_cp, Figure 8-1 illustrates that TSO and TSO-pf suffer from superfluous load speculation failures caused by cache replacement ("flushRep"). These evictions have no effect in WMM.

radix: In radix, Figure 8-1 shows that TSO-pf can match the performance of WMM while TSO is much slower. This implies that store prefetch is crucial. Figure 8-2 shows the amount of time that a store is stalled from being issued into ROB due to full store buffer in radix. The time shown in the figure is normalized to the execution time of WMM. As we can see, TSO suffers from significant store issue stalls, which

causes ROB to be empty for long periods of time ("empty" in Figure 8-1). This is because the store miss latency is so long that in TSO the store buffer stays full most of the time. On the contrary, store prefetch helps TSO-pf drain store buffer much more quickly, and its stall time becomes similar to WMM.

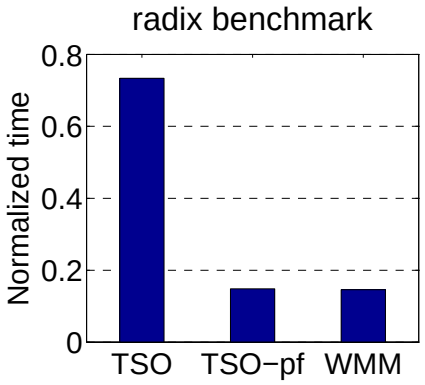

Figure 8-2: Stalls due to full store buffer

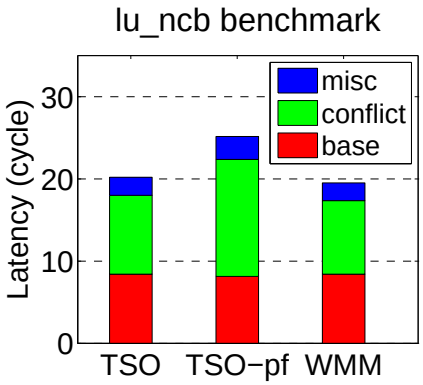

Figure 8-3: Memory read latency

lu_ncb: In lu_ncb, Figure 8-1 shows that TSO can match the performance of WMM while TSO-pf is much slower. To understand this counterintuitive result, we analyze memory read latency in lu_ncb in Figure 8-3. The latency can be broken down into the following three parts:

- **base**: the real processing time. It is the difference between the time when a request accesses L1 tag array, and the time when the response is sent back to the load queue.
- **conflict**: a new request is stalled in L1 while another request for the same cache line is being processed.
- **misc**: all other time spent in memory.

Figure 8-3 shows that TSO-pf has longer memory read latency than TSO and WMM due to the increase in conflict stall time. This counterintuitive result can be understood when we look at the kernel of lu_ncb:

```
for(i=0; i<n; i++) a[i] += alpha * b[i];
```

The prefetch request for store `a[i]` may be on the same cache line as the load of `a[i']`, increasing the number of conflict stalls in L1. That is, store prefetch interferes with load execution and downgrades performance.

Summary: The above analysis reveals two disadvantages of TSO compared to WMM. First, load speculation in TSO is subject to L1 cache eviction. Second, TSO requires store prefetch to reduce store miss latency while store prefetch may sometimes degrade performance due to interference with load execution. In contrast, WMM can perform load speculation without getting affected by L1 cache eviction, and can efficiently hide store miss latency without prefetching.

Chapter 9

Conclusion

In the thesis we have presented WMM, a resilient weak memory model, which is easy to understand because of monolithic memory and *instantaneous instruction execution* (I^2E) style definition. I^2E style definition itself was made possible because of the introduction of a novel conceptual entity called *invalidation buffer*. We have shown that WMM admits all instruction reorderings except stores overtaking loads, and is resilient to speculative microarchitectural optimizations such as memory dependency speculation and load-value prediction. Preliminary evaluation using SPLASH benchmarks shows that WMM implementation outperforms the aggressive implementations of SC and TSO.

Bibliography

[1] Splash-2x benchmarks. `http://parsec.cs.princeton.edu/parsec3-doc.htm#splash2x`.

[2] A memo on exploration of splash-2 input sets. `http://parsec.cs.princeton.edu/doc/memo-splash2x-input.pdf`, 2011.

[3] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[4] Sarita V Adve and Mark D Hill. Weak ordering a new definition. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 2–14. ACM, 1990.

[5] Jade Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012.

[6] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 13–24. ACM, 2009.

[7] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *Programming Languages and Systems*, pages 512–532. Springer, 2013.

[8] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). *Formal Methods in System Design*, 40(2):170–205, 2012.

[9] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):7, 2014.

[10] Ehsan K Ardestani and Jose Renau. Esesc: A fast multicore simulator using time-based sampling. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 448–459. IEEE, 2013.

[11] ARM. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. 2013.

[12] Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 29–40. IEEE Computer Society, 2006.

[13] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *ACM SIGPLAN Notices*, volume 46, pages 55–66. ACM, 2011.

[14] Colin Blundell, Milo MK Martin, and Thomas F Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 233–244. ACM, 2009.

[15] Hans-J Boehm and Sarita V Adve. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[16] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The java memory model: Operationally, denotationally, axiomatically. In *Programming Languages and Systems*, pages 331–346. Springer, 2007.

[17] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: bulk enforcement of sequential consistency. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 278–289. ACM, 2007.

[18] George Z Chrysos and Joel S Emer. Memory dependence prediction using store sets. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 142–153. IEEE Computer Society, 1998.

[19] Edsger W Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[20] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 434–442. IEEE Computer Society Press, 1986.

[21] Walid J Ghandour, Haitham Akkary, and Wes Masri. The potential of using dynamic information flow analysis in data value prediction. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 431–442. ACM, 2010.

[22] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[23] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26. ACM, 1990.

[24] Chris Gniady and Babak Falsafi. Speculative sequential consistency with little custom storage. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 179–188. IEEE, 2002.

[25] James R Goodman. *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991.

[26] Dibakar Gope and Mikko H Lipasti. Atomic sc for simple in-order processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 404–415. IEEE, 2014.

[27] Chris Guiady, Babak Falsafi, and Terani N Vijaykumar. Is sc+ ilp= rc? In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 162–171. IEEE, 1999.

[28] IBM. *Power ISA, Version 2.07*. 2013.

[29] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[30] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient sequential consistency via conflict ordering. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 273–286. ACM, 2012.

[31] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. Value locality and load value prediction. *ACM SIGOPS Operating Systems Review*, 30(5):138–147, 1996.

[32] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[33] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo MK Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for power multiprocessors. In *Computer Aided Verification*, pages 495–512. Springer, 2012.

[34] Jan-Willem Maessen and Xiaowei Shen. Improving the java memory model using crf. *ACM SIGPLAN Notices*, 35(10):1–12, 2000.

[35] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[36] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the arm and power relaxed memory models. `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`, 2012.

[37] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[38] Arthur Perais and André Seznec. Eole: Paving the way for an effective implementation of value prediction. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 481–492. IEEE, 2014.

[39] Arthur Perais and André Seznec. Practical data value speculation for future high-end processors. In *International Symposium on High Performance Computer Architecture*, pages 428–439, 2014.

[40] Parthasarathy Ranganathan, Vijay S Pai, and Sarita V Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 199–210. ACM, 1997.

[41] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising c/c++ and power. In *ACM SIGPLAN Notices*, volume 47, pages 311–322. ACM, 2012.

[42] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *ACM SIGPLAN Notices*, volume 46, pages 175–186. ACM, 2011.

[43] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[44] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile and fences (crf): A new memory model for architects and compiler writers. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 150–161. IEEE, 1999.

[45] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 524–535. IEEE Computer Society, 2012.

[46] SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., 1992.

[47] David L Weaver and Tom Gremond. *The SPARC architecture manual (Version 9)*. PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

[48] Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 266–277. ACM, 2007.

[49] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995.